Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance

Md Kamruzzaman    Steven Swanson    Dean Tullsen
Single Thread Execution

• Single thread performance is important
  – Parallelization is hard

• Amdahl’s Law
  – Serial part becomes dominant as you scale more
Single Thread Execution

- Single thread performance is important
  - Parallelization is hard
- Amdahl’s Law
  - Serial part becomes dominant as you scale more
Single Thread Execution

• Single thread performance is important
  – Parallelization is hard

• Amdahl’s Law
  – Serial part becomes dominant as you scale more

• Architecture trend
  – Becoming more parallel
  – Adding more cores does not help single thread?
Memory Intensive Single Thread Execution

\[
\text{for } i = 1 \text{ to } 100 \{ \\
\quad \text{for } j = 1 \text{ to } 1000 \quad \text{Loop1} \\
\quad \quad a[j] = a[j-1] + a[j+1] \\
\quad \ldots \\
\quad \text{for } j = 1 \text{ to } 2000 \quad \text{Loop2} \\
\quad \quad b[j] = b[j-1] + b[j+1] \\
\}\]
Memory Intensive Single Thread Execution

\[
\text{for } i=1 \text{ to } 100 \{ \\
\quad \text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \\
\quad \quad a[j] = a[j-1] + a[j+1] \\
\quad \quad \text{..................} \\
\quad \quad \text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \\
\quad \quad \quad b[j] = b[j-1] + b[j+1] \\
\}\]
Memory Intensive Single Thread Execution

\[\text{for } i = 1 \text{ to } 100 \{ \]
\[\quad \text{for } j = 1 \text{ to } 1000 \quad \text{Loop1} \]
\[\quad \quad a[j] = a[j-1] + a[j+1] \]
\[\quad \}
\]

\[\text{for } j = 1 \text{ to } 2000 \quad \text{Loop2} \]
\[\quad \quad b[j] = b[j-1] + b[j+1] \]

- One Cache is not large enough to hold both ‘a’ and ‘b’
  - Capacity misses
Can We Exploit Idle Resources?

- Parallel execution in hardware is not always possible
  - Lack of parallelism

```plaintext
for i=1 to 100 {
    for j=1 to 1000  // Loop1
        a[j] = a[j-1] + a[j+1]
    ...........................................
    for j=1 to 2000  // Loop2
        b[j] = b[j-1] + b[j+1]
}
```
Can We Exploit Idle Resources?

for $i=1$ to $100$ {
  for $j=1$ to $1000$  \textbf{Loop1}
    \begin{align*}
    a[j] &= a[j-1] + a[j+1] \\
    \end{align*}
\}

\begin{align*}
\text{for } j=1 \text{ to } 2000 \quad \textbf{Loop2} \\
  b[j] &= b[j-1] + b[j+1] \\
\end{align*}

- Serial execution but \textit{aggregating idle cache resources}
Can We Exploit Idle Resources?

Serial execution but *aggregating idle cache resources*

- **Migrate** computation to another core to use it’s private cache

```plaintext
for i=1 to 100 {
    for j=1 to 1000  
        Loop1
        a[j] = a[j-1] + a[j+1]
    .................
    for j=1 to 2000  
        Loop2
        b[j] = b[j-1] + b[j+1]
}
```
Outline

• Motivation
• **Software Data Spreading**
• Results
• Conclusion
Software Data Spreading

- Migrate **loops** in a single thread computation from core to core in a controlled manner to aggregate idle caches
  - **Parallel type speedup** even for code with no inherent parallelism
  - Saves **power** and energy
  - **Software-only** technique and works on a wide range of machines

- Our compilation framework **automatically** transforms single thread code to apply ‘Data Spreading (DS)’
Software Data Spreading

\[
\text{for } i=1 \text{ to } 100 \{ \\
\quad \text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \\
\quad \quad a[j] = a[j-1] + a[j+1] \\
\quad \} \\
\]

\[
\text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \\
\quad b[j] = b[j-1] + b[j+1] \\
\}

Cache - 0  
Cache - 1  
Cache - 2  
Cache - 3
Software Data Spreading

\[
\text{for } i=1 \text{ to } 100 \{ \\
\quad \text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \\
\quad a[j] = a[j-1] + a[j+1] \\
\}
\]

\[
\text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \\
\quad b[j] = b[j-1] + b[j+1]
\]

Iteration 1 – Spread ‘a[0:1001]’
Software Data Spreading

\[
\text{for } i=1 \text{ to } 100 \{ \\
\text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \\
\quad \quad a[j] = a[j-1] + a[j+1] \\
\} \\
\text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \\
\quad \quad b[j] = b[j-1] + b[j+1] \\
\}
\]

Iteration 1 – Spread ‘\textit{a[0:1001]}’
Software Data Spreading

\[
\text{for } i=1 \text{ to } 100 \{ \\
\text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \\
\quad a[j] = a[j-1] + a[j+1] \\
\}
\]

\[
\text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \\
\quad b[j] = b[j-1] + b[j+1] \\
\}

Iteration 1 – Spread \text{‘}a[0:1001]\text{’}
Software Data Spreading

```plaintext
for i=1 to 100 { 
    for j=1 to 1000 \textbf{Loop1}
    a[j] = a[j-1] + a[j+1]

    \textbf{for j=1 to 2000 Loop2}
    b[j] = b[j-1] + b[j+1]
}
```

Iteration 1 – ‘a[0:1001]’ spread across 4 caches
Software Data Spreading

\[\text{for } i=1 \text{ to } 100 \{\]

\[\text{for } j=1 \text{ to } 1000 \quad \text{Loop 1}\]
\[a[j] = a[j-1] + a[j+1]\]

\[\text{for } j=1 \text{ to } 2000 \quad \text{Loop 2}\]
\[b[j] = b[j-1] + b[j+1]\]

\}\]

Iteration 1 – ‘\(b[0:2001]\)’ spread across 4 caches
Software Data Spreading

\[ \text{for } i=1 \text{ to } 100 \{ \]
\[ \text{for } j=1 \text{ to } 1000 \text{ Loop1} \]
\[ a[j] = a[j-1] + a[j+1] \]

\[ \text{for } j=1 \text{ to } 2000 \text{ Loop2} \]
\[ b[j] = b[j-1] + b[j+1] \]

\}\]

'\text{a[0:251]}' in the cache

\text{Iteration 2 – Hits in private cache}
Software Data Spreading

\[ \text{for } i=1 \text{ to } 100 \{ \]

\[ \text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \]
\[ a[j] = a[j-1] + a[j+1] \]

\[ \text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \]
\[ b[j] = b[j-1] + b[j+1] \]

\} \quad \text{Computation Follows the Data}

Iteration 2 – Hits in private cache
Software Data Spreading

\[ \text{for } i=1 \text{ to } 100 \{ \text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \]
\[ a[j] = a[j-1] + a[j+1] \]

\[ \text{.........................} \]
\[ \text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \]
\[ b[j] = b[j-1] + b[j+1] \]
\[ \} \]

• No Data Spreading
  – 98% L2 Miss rate

• With Data Spreading
  – 2% L2 Miss rate
How to Apply Data Spreading

• Computation migrates to ‘*where the required data exists’*
  – Predictable memory behavior
  – Loops are natural targets

```
for(...)  
  for(...)  
    for(...)  
      for(...)  
       for(...)  
```

• Can we spread any set of loops?
  – Not all loops are ideal candidates.
How to Apply Data Spreading

- Computation migrates to ‘where the required data exists’
  - Predictable memory behavior
  - Loops are natural targets

```plaintext
for(...) 
  for(...) 
    for(...) 
      for(...) 
        for(...) 
```

Loop Nest Tree

Which loops are ideal candidates for spreading? How do we spread the candidate loops?
Which Loops are Ideal Candidates

- **Epoch**: The execution of a loop instance
- **Epoch Sharing (ES)**: A measure of data sharing between different epochs of a loop

![Graph showing types of loops for ideal candidates]

- 0% ES
- 50% ES
- 100% ES

**Memory Region**

**Bad**

**Good**
Which Loops are Ideal Candidates

- **Epoch**: The execution of a loop instance
- **Epoch Sharing (ES)**: A measure of data sharing between different epochs of a loop

High ES is Common

<table>
<thead>
<tr>
<th>Memory Region</th>
<th>Bad</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>0% ES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50% ES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100% ES</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Which Loops are Ideal Candidates

```plaintext
for i=1 to 100 {
    for j=1 to 1000 Loop1
        a[j] = a[j-1] + a[j+1]
    }
```

- Profile to construct a dynamic loop nest tree
  - Epoch sharing
  - Maximum epoch foot print
    - Size of the largest memory footprint over all epochs
  - Amortized migration overhead
    - Ratio of the migration cost to the total computation
Which Loops are Ideal Candidates

\[
\text{for } i=1 \text{ to } 100 \{ \\
\text{for } j=1 \text{ to } 1000 \{ \\
\text{ for } j=1 \text{ to } 2000 \{ \\
\text{ ES=100 } \\
\text{ ES=100 } \\
\text{ ES=100 } \\
\text{ Candidate } \\
\text{ Candidate }
\}
\}
\}
\]

- Select a loop as candidate only if it passes the \textit{ES, max epoch footprint} and \textit{migration overhead} thresholds
  - All 3 loops pass the test

- Why not pick Loop0?

\[
\text{for } i=1 \text{ to } 100 \{ \\
\text{ for } j=1 \text{ to } 1000 \{ \\
\text{ for } j=1 \text{ to } 2000 \{ \\
\text{ a}[j] = a[j-1] + a[j+1] \\
\text{ b}[j] = b[j-1] + b[j+1] \\
\}
\}
\}
\]
Which Loops are Ideal Candidates

```
for i=1 to 100 { Loop0
    for j=1 to 1000 Loop1
        a[j] = a[j-1] + a[j+1]
    }
    for j=1 to 2000 Loop2
        b[j] = b[j-1] + b[j+1]
}
```

- Inner and outer loop spreading conflict
  - Migrations in the inner loop render outer loop spreading useless
- Spreading the loop does not spread the data
  - Loop0 touches the same data in every iteration — will not spread ‘a’, ‘b’
Which Loops are Ideal Candidates

```
for i=1 to 100 {  
  for j=1 to 1000  
  a[j] = a[j-1] + a[j+1]
  ................................
  for j=1 to 2000  
  b[j] = b[j-1] + b[j+1]
}
```

- Inner and outer loop spreading conflict
  - Migrations in the inner loop render outer loop spreading useless
- Spreading the loop does not spread the data
  - Loop0 touches the same data in every iteration — will not spread ‘a’, ‘b’
- Bottom up Algorithm
  - If a loop is selected, none of its ancestors are selected
How to Spread Candidate Loops

\[
\text{for } i=1 \text{ to } 100 \{ \quad \text{Loop0} \\
\quad \text{for } j=1 \text{ to } 1000 \quad \text{Loop1} \\
\quad \quad a[j] = a[j-1] + a[j+1] \\
\quad \ldots \\
\quad \text{for } j=1 \text{ to } 2000 \quad \text{Loop2} \\
\quad \quad b[j] = b[j-1] + b[j+1] \\
\}
\]

- Split a candidate loop **evenly** into \( K \) (#of caches) parts
- Migrate to next core once current part is executed
How to Spread Candidate Loops

for $i=1$ to $100$  
  for $j=1$ to $1000$  
    $a[j] = a[j-1] + a[j+1]$  
  
for $j=1$ to $2000$  
  $b[j] = b[j-1] + b[j+1]$  

• Split a candidate loop \textbf{evenly} into $K$ (#of caches) parts
• Migrate to next core once current part is executed
Results
Evaluation

- Memory intensive benchmarks from NAS, Spec2000, Spec2006, and micro-benchmarks

- Evaluated on a variety of real machines
  - Quad socket Pentium 4
  - Dual socket Core2Quad
  - Dual socket Nehalem
  - Dual socket Opteron
Results – Dual Socket Core2Quad

WS fits in L2

WS does not fit in L2

Working Set (MB)

6M L2 6M L2 6M L2 6M L2
Results – Dual Socket Core2Quad

7X
WS fits in combined L2

12X

<table>
<thead>
<tr>
<th>Working Set (MB)</th>
<th>6M L2</th>
<th>6M L2</th>
<th>6M L2</th>
<th>6M L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory operations per µs</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq-Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq-DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rand-Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rand-DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results – Dual Socket Core2Quad

- WS does not fit in combined L2

![Graph showing Memory operations per µs vs Working Set (MB)]
Results – Dual Socket Core2Quad

WS does not fit in combined L2

Working Set (MB)

6M L2  6M L2  6M L2  6M L2
Results – Dual Socket Core2Quad

- **Nehalem** – 3X, **Opteron** – 4X and **Pentium4** – 5X

WS does not fit in combined L2
Dual Socket Opteron Cache Hierarchy

Socket 1
- 6M L3
- L2
- L2
- L2
- L2
- L2

Socket 2
- 6M L3
- L2
- L2
- L2
- L2
- L2

512 K
Data Spreading – Sockets vs Cores

Opteron – DS across sockets
Data Spreading – Sockets vs Cores

Opteron – DS across sockets

Opteron – DS across cores
Core2Quad – Throughput vs Speedup

- Throughput converts into speedup
- Memory intensive applications get more benefit
2D Jacobi – Speedup Across Machines

- Speedup depends on the interplay between the working set size and the aggregate cache space
• Working set sizes vary a lot
Benchmarks – Speedup Across Machines

- Speedup can be significant – Art gets 2.2X in Pentium 4
Benchmarks – Speedup Across Machines

- Speedup can be significant – Art gets 2.2X in Pentium 4
- Working set and aggregate cache space interaction
  - Pentium 4 works better for smaller ‘A’ input of LU
  - Other machines work better for larger ‘B’ input
• DS eliminates as much as **80%** of the cache misses
• Migration cost, loosing current cache state and costly cache to cache transfers limit the effectiveness
• Core2Quad
  – Random access pointer chasing microbenchmark
**Power Consumption**

- **Saves power** by eliminating accesses to DRAM
  - For Core2Quad up to 51W or 81% of non-idle system power
- **Energy savings even greater**
Conclusion

• **Data Spreading** is a *software only* technique that aggregates idle cache resources for single thread

• When the working set fits in the aggregate cache space, performance gain is huge (as large as 12X)

• Data spreading also *saves power (and energy)*

• Our compilation framework *automatically* transforms single thread code to apply data spreading
Thank You

Questions ?
Backup Slides
High ES is Common

The chart illustrates the percentage of loops for different benchmarks with various performance metrics. Each bar represents a benchmark (Art, Applu, Equake, Mcf, Swim, Libq, BT, CG, LU, MG, SP) and the segments within the bar indicate the proportion of loops with different execution speeds (ES). The chart shows that a significant portion of loops exhibit high ES, indicating that high performance is common.
Migration Technique

- **OS Directed Technique**
  - Starting from Linux Kernel 2.6
  - `sched_setaffinity` pins a thread to a core
  - Average latency 9-14 $\mu$s
  - Only one thread remains active

- **Userspace Thread Migration Technique**
  - No system call
  - Multiple threads are active – one executing while others spinning
  - Average latency 1-3 $\mu$s
Core2Quad – User CS Performance
Nehalem CMP – Microbenchmark
Opteron CMP – Benchmarks
## Candidate Loops

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Total Loops</th>
<th>Candidate Loops</th>
<th>% Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art</td>
<td>84</td>
<td>20</td>
<td>24%</td>
</tr>
<tr>
<td>Applu</td>
<td>199</td>
<td>12</td>
<td>6%</td>
</tr>
<tr>
<td>Equake</td>
<td>121</td>
<td>6</td>
<td>5%</td>
</tr>
<tr>
<td>Mcf</td>
<td>70</td>
<td>9</td>
<td>13%</td>
</tr>
<tr>
<td>Swim</td>
<td>69</td>
<td>9</td>
<td>13%</td>
</tr>
<tr>
<td>Libq</td>
<td>63</td>
<td>14</td>
<td>22%</td>
</tr>
<tr>
<td>BT</td>
<td>243</td>
<td>43</td>
<td>18%</td>
</tr>
<tr>
<td>CG</td>
<td>63</td>
<td>14</td>
<td>22%</td>
</tr>
<tr>
<td>LU</td>
<td>190</td>
<td>15</td>
<td>8%</td>
</tr>
<tr>
<td>MG</td>
<td>82</td>
<td>6</td>
<td>7%</td>
</tr>
<tr>
<td>SP</td>
<td>333</td>
<td>72</td>
<td>22%</td>
</tr>
</tbody>
</table>
Epoch Sharing (ES)

\[ ES(l) = \frac{\sum_{i=0}^{k-1} S(\bigcup_{j<i} e_j, e_i)}{k-1} \]

\[ S(e_i, e_j) = \frac{|e_i \cap e_j| \times 100}{|e_i \cup e_j|} \]

\( e_i \) denotes the memory footprint of \( i^{th} \) epoch
Latency Variations

- Significant latency gap between different levels of memory hierarchy
  - 3 level cache hierarchy in AMD Opteron
- Converting Last Level Cache (LLC) misses into hits saves a lot
Results – Core2Quad L2 Misses

- LU – Converts 80-90% of L2 misses into L2 hits
- 11 out of 21 benchmarks, data spreading eliminates 20% of misses or more